A Branch-and-Bound Algorithm for the Coupled Task Problem

József Békési*, Gábor Galambos*, Michael N. Jung†,
Marcus Oswald†, Gerhard Reinelt†

April 3, 2013

Abstract

The coupled task problem is to schedule jobs on a single machine where each job consists of two subtasks and where the second subtask has to be started after a given time interval with respect to the first one. The problem has several applications and is NP-hard. In this paper we present a branch-and-bound algorithm for this problem and compare its performance with four integer programming models.

1 The Coupled Task Problem

The *Coupled Task Problem (CTP)* deals with scheduling n jobs each of which consists of two subtasks and where there is the additional requirement that between the execution of these subtasks an exact delay is required. Such jobs are sometimes also called *interleaving two-phase jobs* in the literature. We use the following notation. The set of jobs is $\{J_1, J_2, \ldots, J_n\}$ and the set of tasks is $\{T_1, T_2, \ldots, T_{2n}\}$, where T_{2i-1} and T_{2i} denote the first and second subtask of J_i. For improving readability when distinguishing between the first and the second task of job J_i, the index $2i-1$ is denoted by i_A and the index $2i$ by i_B. We will usually refer to jobs or tasks just by their index. The processing times for the first and second task of job J_i are a_i and b_i resp., and the delay between the tasks is l_i. The processing time for task T_k is denoted p_k. The *system time* time for job J_i is denoted $t_i = a_i + l_i + b_i = p_{2i-1} + l_i + p_{2i}$. All processing times and delays are assumed to be positive integer numbers.

A single machine is available to process the jobs. They may be scheduled in any order with the restriction that no two tasks occupy the machine at the same time. Interrupting a task and resuming it later (*preemption*) is not allowed. The objective is to minimize the makespan of the schedule, i.e., to minimize the termination time of the job finished last. This is equivalent to minimizing the total time where the processor is idle. Idle times occur between the execution of tasks and we will speak about *gaps* in the schedule. For an easy distinction we call the requested gap between the execution of the two tasks of a job a *break*.

*Department of Computer Sciences, Juhász Gyula Teacher Training Faculty, University of Szeged, Boldogasszony sgt. 3, H-6701 Szeged, Hungary
Supported by the European Union and the European Social Fund through project "Supercomputer, the national virtual lab", grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0010
†Institut für Informatik, Fakultät für Mathematik und Informatik, Universität Heidelberg, Im Neuenheimer Feld 368, D-69120 Heidelberg, Germany
Supported by DAAD exchange program 324 PPP-Ungarn

The CTP was introduced and shown to be NP-hard in [6]. Example applications are the management of a radar station or the operation of airplane carriers [7]. By imposing restrictions on the parameters a_i, b_i and l_i several special cases of the CTP have been investigated. Most of them remain NP-hard, but a few can be solved in polynomial time [3, 10, 5]. The special case where $a_i = a$, $l_i = l$, and $b_i = b$, for $i = 1, \ldots, n$, is called *Identical Coupled Task Problem (IdCTP)*. It is of particular interest because its complexity is still open. Further generalizations of the problem have been studied, e.g., [8] considers the case where the breaks are not fixed but have to satisfy lower and upper bounds.

In this paper we focus on the optimal solution of the general coupled task problem with fixed breaks and without any additional assumptions on the parameters. In Section 2, we present a new branch-and-bound algorithm which is of combinatorial nature and, in particular, does not make use of linear programming. We compare it with four linear integer models described in Section 3: a time-indexed model, two models based on linear ordering variables, and a model given by Sherali and Smith in [9]. Computational results on various types of coupled task problem instances are presented in Section 4. Some conclusions are given in the final Section 5.

2 The branch-and-bound approach

The general coupled task problem or variants of it have so far been approached by integer programming or dynamic programming algorithms [2, 4, 9]. As an alternative and also for comparison with integer programming approaches for the general problem we designed a branch-and-bound algorithm where bounds are computed in a purely combinatorial way without the need of linear programming. Its main special features are the way of defining the subproblems and the various lower bound computations.

Generation of subproblems

As usual in a branch-and-bound algorithm we first try to solve the given problem instance by computing a feasible solution yielding an upper bound on the optimum value and by exhibiting a lower bound obtained by certain bounding procedures (described below). If the two bounds coincide, then the problem is solved to proven optimality. Otherwise we replace the instance by a collection of subproblems such that the union of their feasible solutions contains all feasible solutions. Then it is tried to solve these subproblems and, if some cannot be solved, the process is iterated. A subproblem can also considered solved if a lower bound for its feasible solutions is computed which is larger than or equal to the value of the best known feasible solution of the original problem.

In our approach subproblems are created in a special way by requiring that some linear ordering has to be satisfied for a subset of tasks. The root node of the branch-and-bound tree (corresponding to the original problem) is specified by the empty ordering, i.e., there is no restriction on the sequence of tasks. Consider a subproblem with associated ordering π for some tasks. If the problem cannot be solved, child nodes are constructed by augmenting π by inserting a new job i obeying the following rule: the tasks of this job may only be inserted after the last task of π which is the first subtask of a job. Note that a subproblem is only characterized by an order in which part of the tasks have to be executed and that no concrete start times are specified yet.

Let L_A denote the last task of π which is the first subtask of a job and L_B denote the corresponding second task. The rule for creating nodes implies that the order π will not be changed up to and including L_A when creating subproblems. The children of a node are in principle all possible partial orderings obtained by extending π with a not yet included job i, and inserting its first task i_A at any position after L_A. For the second task i_B we just assume that it is positioned after i_A. So, from the original problem we can possibly generate n subproblems on level 1 of the branch-and-bound tree. Each of these subproblems can possibly lead to $3(n-1)$ subproblems on level 2, and so on. Thus, at first sight, the branch-and-bound tree can grow very fast, but it will turn out that many of the potential child nodes are infeasible. And by excluding some symmetries (see below) the size of the tree can be reduced further.

We call a subsequence of a partial ordering a *block* if no further insertion of tasks into the sequence is possible due to the extension rule and if for every job either both tasks are contained in it or none of them. A block is called *active* if it may be extended with further jobs. Obviously, at any time there is only one active block. Namely, if when creating a child node both tasks of the new job are placed at the end of the partial ordering π, then the previously active block is considered completed and a new active block is created. For all other placements of the new job the active block remains active. Figure 1 illustrates the block structure resulting from the placement of tasks. The left block is completed because no other task may be inserted and for every jobs its two tasks are present. The active block (containing L_A) consists of four task and there are three insertion possibilities into the corresponding ordering.

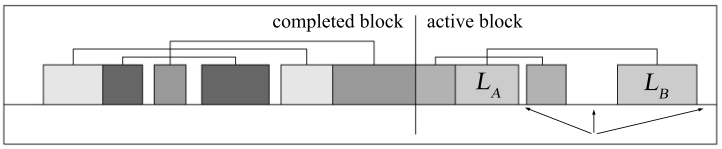

Figure 1: Illustration of the block structure and task placement.

Feasibility test

For checking the feasibility of a partial ordering π, we calculate lower and upper bounds (a *time window*) for the start time of each of its tasks. The lower bound for the first task is initialized with 0 and the upper bound for the last task is initialized with the current global upper bound minus 1 minus its processing time, since with any later placement the solution would not lead to a solution better than the currently known best one. Then, the bounds for all tasks in the partial ordering are updated iteratively by taking the bounds of their predecessors and successors as well as the bounds of the other task of the same job into account. This is done until no further improvement of bounds is possible. A potential node is infeasible or can be pruned if the lower bound on the start time of one of its tasks exceeds its upper bound, i.e., its time window becomes empty. Otherwise the associated partial ordering can potentially lead to a better solution.

Algorithm 1: Feasibility test

Input: partial ordering $\pi = (\pi_1, \ldots, \pi_{2k})$, best known solution value UB
Output: feasibility and, if feasible, time windows $(lb_{\pi_1}, ub_{\pi_1}), \ldots, (lb_{\pi_n}, ub_{\pi_n})$
begin
  `// Initialization of bounds on starting times`
  $lb_{\pi_1} = 0,\ ub_{\pi_1} = 0$
  for $i = 2, \ldots, 2k$ **do**
    $lb_{\pi_i} = p_{\pi_1}$
    $ub_{\pi_i} = UB - p_{\pi_{2k}} - 1$
  $lb_{s(\pi_1)} = a_{d(\pi_1)} + l_{d(\pi_1)},\ ub_{s(\pi_1)} = a_{d(\pi_1)} + l_{d(\pi_1)}$
  $ub_{f(\pi_{2k})} = ub_{\pi_{2k}} - a_{d(\pi_{2k})} - l_{d(\pi_{2k})}$
  `// Start of loop that calculates time windows`
  while *there are bound changes* **do**
    `// Adjust lower bounds`
    for $i = 2, \ldots, n$ **do**
      if $lb_{\pi_i} < lb_{\pi_{i-1}} + p_{\pi_{i-1}}$ **then**
        $lb_{\pi_i} = lb_{\pi_{i-1}} + p_{\pi_{i-1}}$
        if $\pi_i \in F$ **then**
          $lb_{s(\pi_i)} = lb_{\pi_i} + a_{d(\pi_i)} + l_{d(\pi_i)}$
        else
          $lb_{f(\pi_i)} = lb_{\pi_i} - a_{d(\pi_i)} - l_{d(\pi_i)}$
    `// Adjust upper bounds`
    for $i = n-1, \ldots, 1$ **do**
      if $ub_{\pi_i} > ub_{\pi_{i+1}} - p_{\pi_{i+1}}$ **then**
        $ub_{\pi_i} = ub_{\pi_{i+1}} - p_{\pi_{i+1}}$
        if $\pi_i \in F$ **then**
          $ub_{s(\pi_i)} = ub_{\pi_i} + a_{d(\pi_i)} + l_{d(\pi_i)}$
        else
          $ub_{f(\pi_i)} = ub_{\pi_i} - a_{d(\pi_i)} - l_{d(\pi_i)}$
    `// Test feasibility`
    for $i = 1, \ldots, n$ **do**
      if $lb_{\pi_i} > ub_{\pi_i}$ **then**
        Return *infeasible*.
  Return *feasible* and the time windows.

The basic feasibility test is given as Algorithm 1. For ease of notation we use the functions $f : S \to F$ mapping a second task to its corresponding first task, $s : F \to S$ mapping a first task to its corresponding second task, and $d : \{\pi_1, \ldots, \pi_k\} \to \{1, \ldots, n\}$ associating the corresponding job numbers with tasks of the ordering.

Note that the bounds of the tasks belonging to the same job are always adapted simultaneously and hence the placement of the two corresponding tasks always remains valid with respect to the break l_i. As a byproduct, this feasibility test produces time windows for the start times of each task in π that can be exploited for a better bound estimation in the subsequent bounding heuristics.

Note that, when a new active block is created, all tasks of the previously active block can be started at the lower bounds of their time windows. This is valid because there will be no future interaction with the completed block and every start time within the window of these tasks is feasible. (Of course, the starting times of coupled tasks have to observe the break.) Starting times fixed this way can also be kept for child nodes, so the update of time windows only needs to be done for the currently active block. Therefore, at child nodes, Algorithm 1 only has to be carried out for the partial ordering representing the active block and the time window of the first task of it can be started with the completion time of the previously completed block.

Bound calculation

Assume, we are at a node of depth k with partial ordering $\pi = (\pi_1, \ldots, \pi_{2k})$ and let Π be the set of jobs in π. We compute lower bounds on the makespan of child nodes by taking also tasks into account which are not yet inserted. Note, that the time windows computed in the feasibility test can be exploited. We assume that the tasks L_A and L_B are defined as above and belong to job m with positions $\pi_A = L_A$ and $\pi_B = L_B$.

Four bounding procedures of increasing computational effort are employed. It will turn out that these bounds can be computed very fast and many nodes can be explored (and possibly discarded) in a reasonable amount of time. The heuristics try to exploit different properties of the jobs, e.g., heuristic $B4$ is very helpful if some jobs cannot intertwine with others or if system times of jobs are very large.

These four bounding procedures are executed in order $B1$–$B4$. The order is chosen such that the heuristics become more and more time consuming. A bound computation has to be applied only if the previous bound is still smaller than the current best solution, because otherwise the node can be pruned.

$B1$ Trivial bound

Since we know that the children of π do not insert a job before L_A, we can compute a trivial lower bound by summing up the lower bound for the finishing time of L_A, the processing times of all tasks in π scheduled after L_A, and the processing times of all tasks not yet included in π. Thus the bound is

$$B1 = lb_{L_A} + p_{L_A} + \sum_{i=A+1}^{2k} p_{\pi_i} + \sum_{j \notin \Pi} (a_j + b_j).$$

For the root node this is just the sum of the processing times of all tasks.

Algorithm 2: $B2$ Analysis of the break between L_A and L_B

Input: ordering $\pi = (\pi_1, \ldots, \pi_{2k})$ and time windows $(lb_{\pi_1}, ub_{\pi_1}), \ldots, (lb_{\pi_n}, ub_{\pi_n})$
Output: lower bound $B2$ on the makespan of a schedule which respects π
begin

```
// Initialization of bound and compute how much of the break is left
```

$\mathcal{P}_F = \emptyset$
$\mathcal{P}_N = \emptyset$
$g_{B2} = l_m - \sum_{j=A+1}^{B-1} p_{\pi_j}$
$B2 = lb_{L_B} + p_{L_B} + \sum_{j=B+1}^{2k} p_{\pi_j}$

```
// Test how not inserted jobs fit into the break g if started after
   L_A
```

for $j = 1, \ldots, n$ **do**
  if $j \notin \Pi$ **then**
    `// Simple test if it could not fit completely`
    if $a_j + b_j > g_{B2}$ *or* $t_j > l_m$ **then**
      `// Test if only first could fit`
      if $a_j \leq g_{B2}$ *and* $l_j \geq b_m$ **then**
        $B2 = B2 + b_j$
        $\mathcal{P}_F = \mathcal{P}_F \cup \{j\}$
      else
        $B2 = B2 + a_j + b_j$
        $\mathcal{P}_N = \mathcal{P}_N \cup \{j\}$

Return $B2$

$B2$ Analysis of the break between L_A and L_B

This procedure tries to identify tasks which cannot be placed before L_B. The processing times of these tasks can then be added to the finishing time of L_B.

Obviously, all tasks that are already placed after L_B in the partial ordering belong to this set. For the jobs which are not yet in π Algorithm 2 is executed. It tries to place the tasks of these jobs between L_A and L_B taking into account which share of the break between L_A and L_B is already assigned for other tasks. The sets $\mathcal{P}_F$ and $\mathcal{P}_N$ are used to collect the indices of jobs, which do not completely fit into the break ($\mathcal{P}_F$ if only the first subtask fits, $\mathcal{P}_N$ if neither subtask fits).

Figure 2 illustrates the tests in the inner loop of Algorithm 2. The upper job, which is not yet included in the partial ordering, does neither completely fit into the break, nor can its tasks be intertwined with the task L_B, hence it belongs to $\mathcal{P}_N$. The lower job could possibly be intertwined with L_B if the white job would be scheduled a little earlier, but the job does not completely fit into the break, hence it would be placed in $\mathcal{P}_F$.

Since we work with time windows for the start times, jobs can be moved in the schedule,

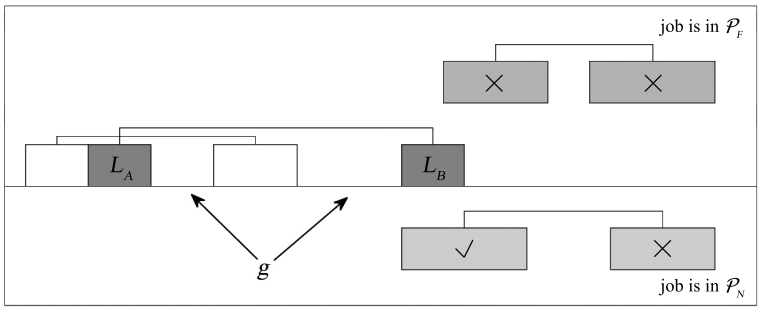

Figure 2: Illustration of the inner loop of Algorithm 2.

as the white job in Figure 2. To take all possible start times into account would be too time consuming for the bounding procedure, so we do not incorporate this at the price of obtaining weaker bounds.

Now the second bound is computed as

$$B2 = lb_{L_B} + p_{L_B} + \sum_{i=B+1}^{2k} p_{\pi_i} + \sum_{j \in \mathcal{P}_N} (a_j + b_j) + \sum_{j \in \mathcal{P}_F} b_j.$$

Procedure B2 is most effective if the tasks have relatively large processing times compared to the break sizes which makes intertwining of jobs rather difficult. Then, the improvement with respect to the trivial bound $B1$ can be expected to be in the range of the gap left between L_A and L_B.

$B3$ Improved analysis of inevitable idle times in gaps

This bound is initialized with the trivial bound $B1$ and, in addition, considers gaps that are already apparent in the partial ordering π. The procedure tries to estimate how good these gaps can be filled with not yet placed jobs and adds the amount that cannot be filled for each gap since this amounts to inevitable idle time of the processor.

The estimation of the best gap fillings can be seen as solving a knapsack problem where the processing times of the tasks not yet scheduled are the item sizes and the gap size is the capacity of the knapsack. However, since solving these knapsack problems exactly is too time consuming, we proceed in a slightly different way. For small gaps we compute optimum solutions, but for larger gaps we relax the problem by allowing the filling of the gap to be composed of all tasks. The latter computations can be performed before starting the branch-and-bound algorithm and the results are stored in a table for future access.

We consider gaps as small if their size is below the four smallest possible gap sizes that could be filled completely with tasks not included in the partial ordering. (The choice of four gaps turned out to be best in our computational experiments.) These four smallest values g_i, $i = 1, \ldots, 4$, can easily be computed as the minimum sum of i processing times of different tasks not yet contained in π. In addition we define $g_0 = 0$. For a gap of size g, let $z(g)$ be the largest of these numbers less than or equal to g, i.e., $z(g) = \max\{g_i \mid g_i \leq g, 0 \leq i \leq 4\}$.

The number $z(g)$ thus is the best filling of the gap g using only the smallest tasks. If $z(g) \neq g_4$, i.e., the best filling is produced by one of the smaller fillings, then the amount $g - z(g)$ can be added to the lower bound since the gap cannot be filled any better and the

Algorithm 3: $B3$ Analysis of idle times in gaps

Input: ordering $\pi = (\pi_1, \ldots, \pi_{2k})$ and time windows $(lb_{\pi_1}, ub_{\pi_1}), \ldots, (lb_{\pi_n}, ub_{\pi_n})$
Output: lower bound $B3$ on the makespan of a schedule which respects π
begin
  `// Initialization of best fillings of 4 smallest gaps`
  compute g_i, $0 \leq i \leq 4$
  $B3 = B1 + g_{B2} - \bar{z}(g_{B2})$
  for $j = B, \ldots, 2k-1$ **do**
    `// calculate variable gap sizes between` π_j `and` π_{j+1}
    $lb = lb_{\pi_{j+1}} - ub_{\pi_j}$
    $ub = ub_{\pi_{j+1}} - lb_{\pi_j}$
    `// for all possible gap sizes calculate best filling and add the minimum remainder of the gap to the bound`
    $B3 = B3 + \min\{i - \bar{z}(i) \mid lb \leq i \leq ub\}$
  Return $B3$

schedule will have at least this idle time. If $z(g) = g_4$, possible fillings other than the smallest four have to be taken into account.

This procedure can be improved by considering also larger gaps. During preprocessing we compute a look-up table which gives for each possible gap size the maximum filling using combinations of any tasks. If the amount of what is left of a gap exceeds the four smallest values, then the look-up table can be used and the difference between the gap size and its best filling can be added as well. In the following, we will refer to the best filling of gap g by all tasks as $Z(g)$ (already calculated during preprocessing for all relevant gap sizes). Then the best filling $\bar{z}(g)$ with respect to the current ordering π is computed as

$$\bar{z}(g) = \begin{cases} z(g) & \text{if } g \leq g_4, \\ Z(g) & \text{otherwise.} \end{cases}$$

The first gap considered is the gap left between L_A and L_B whose size was already computed as g_{B2} in procedure $B2$. If we want to take into account other gaps as well, we have to make sure that they do not overlap. For the procedure, we hence apply the method only to the gaps between consecutive tasks in π appearing after L_B. It has to be noted that the sizes of these gaps are often not yet fixed but can vary. This happens, when at least one of the adjacent tasks has $lb_i < ub_i$. Algorithm 3 describes the procedure in detail.

This procedure is most effective when the processing times of tasks are relatively large which makes it difficult to fill breaks almost completely. This is the case in test sets T3 and T4 (cf. Section 4). The procedure is also effective deeper in the tree when there are only few jobs left to fill the gaps.

$B4$ Analysis of system times of jobs not inserted yet

For the fourth bound each unscheduled job i is considered separately and an individual bound $B4_i$ is computed. The bounds are initialized with the earliest finishing time of L_A. Then the system time of job i is added and all processing times of tasks executed after L_A

Algorithm 4: $B4$ Analysis of system times of jobs not inserted yet

Input: ordering $\pi = (\pi_1, \ldots, \pi_{2k})$ and time windows $(lb_{\pi_1}, ub_{\pi_1}), \ldots, (lb_{\pi_n}, ub_{\pi_n})$
Output: lower bound $B4$ on the makespan of a schedule which respects π
begin

```
// go through not yet inserted jobs
for i ∈ {1,...,n}\Π do
    B4_i = lb_{L_A} + p_{L_A} + t_i
    // Go through the partial ordering after L_A, upper part in
       Figure 3
    for j = A+1,...,2k do
        // If a task in π does not fit into J_i from the left, we can
           add its exectution time
        if l_{d(π_j)} < a_i or l_i < p_{π_j} then
            B4_i = B4_i + p_{π_j}
    // go through other not yet inserted jobs, lower part in Figure 3
    for j ∈ {1,...,n}\(Π ∪ {i}) do
        if J_j fits into the gap of J_i: t_j ≤ l_i then
            δ_ij = 0
        else if they can be intertwined in both ways: a_i, b_i ≤ l_j, a_j, b_j ≤ l_i then
            δ_ij = min{a_j, b_j}
        else if J_i fits into J_j from the right: a_i ≤ l_j, b_j ≤ l_i then
            δ_ij = a_j
        else if J_i fits into J_j from the left: b_i ≤ l_j, a_j ≤ l_i then
            δ_ij = b_j
        else they can not be intertwined at all
            δ_ij = a_j + b_j
        B4_i = B4_i + δ_ij
B4 = max_{i∈{1,...,n}\Π} B4_i
```

which cannot be put in the gap of i. This includes the other jobs not yet placed. Figure 3 illustrates when processing times can be added for a given job i. The checked tasks could fit, whereas the crossed tasks do not fit. For the partial ordering π at the top only the last task could be intertwined with i. The jobs not yet scheduled at the bottom fit as described.

The bound is computed with Algorithm 4. Note that the computation of δ_{ij} in the algorithm can also be executed for all pairs of jobs i and j before the branch-and-bound algorithm is started and the information is just looked up at runtime. However, it is included in the algorithm description to display how it is computed. The same function d as in Algorithm 1 is used.

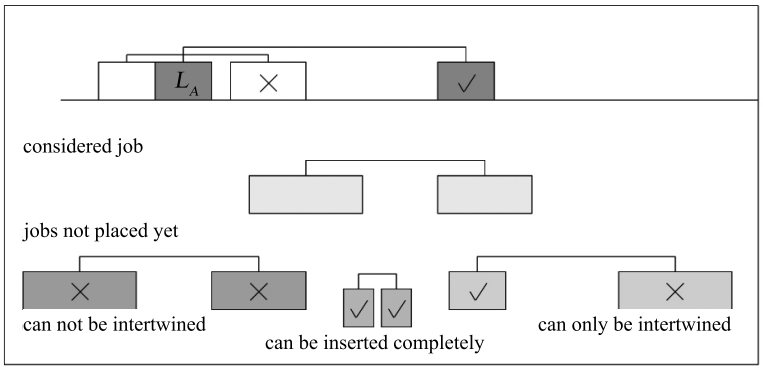

Figure 3: Illustration of procedure B4.

Node selection

A basic decision in the design of a branch-and-bound algorithm concerns the sequence in which the generated subproblems are processed, i.e., the question which of the unprocessed nodes of the branch-and-bound tree is chosen next. Our strategy for node selection can be seen as a combination of *depth-first* and *best-bound-first* search. On the one hand it finds good feasible solutions fast (by diving in a depth-first manner into the tree for reaching promising leaves), on the other hand it tries to compute tight bounds. For the latter goal, the lower bound for each node is computed when the node is created. During the best-bound-first part the node with the lowest lower bound is processed next. This approach is very effective since the bound calculating procedures are very fast.

Symmetry breaking

The success of a branch-and-bound algorithm heavily depends on the possibility to prunes nodes on low levels of the tree. Besides by finding good lower bounds, a node can also be discarded if we can argue that by certain symmetry arguments there is another node which leads to solutions at least as good.

A first elimination of unnecessary nodes is possible by considering blocks in the partial orderings. Since blocks do not overlap the schedule can be calculated for each block individually. For one given solution, all solutions, which are just a permutation of the same blocks, have the same makespan. Therefore, we only accept the permutation in which the blocks are sorted such that the first tasks are sorted lexicographically.

Another observation is that also identical coupled tasks should be ordered lexicographically. This has only a small impact, since this situation rarely occurs, but it only involves negligible computational effort.

Figure 4 visualizes the subproblem generation and shows some effects of the symmetry breaking. The branch-and-bound tree of this example has three jobs where jobs 1 and 3 have identical data. The figure also illustrates how fast the tree can grow in principle.

3 Integer programming models

For assessing the performance of the branch-and-bound algorithm we have implemented four integer programming models for the general coupled task problem. Except for the model

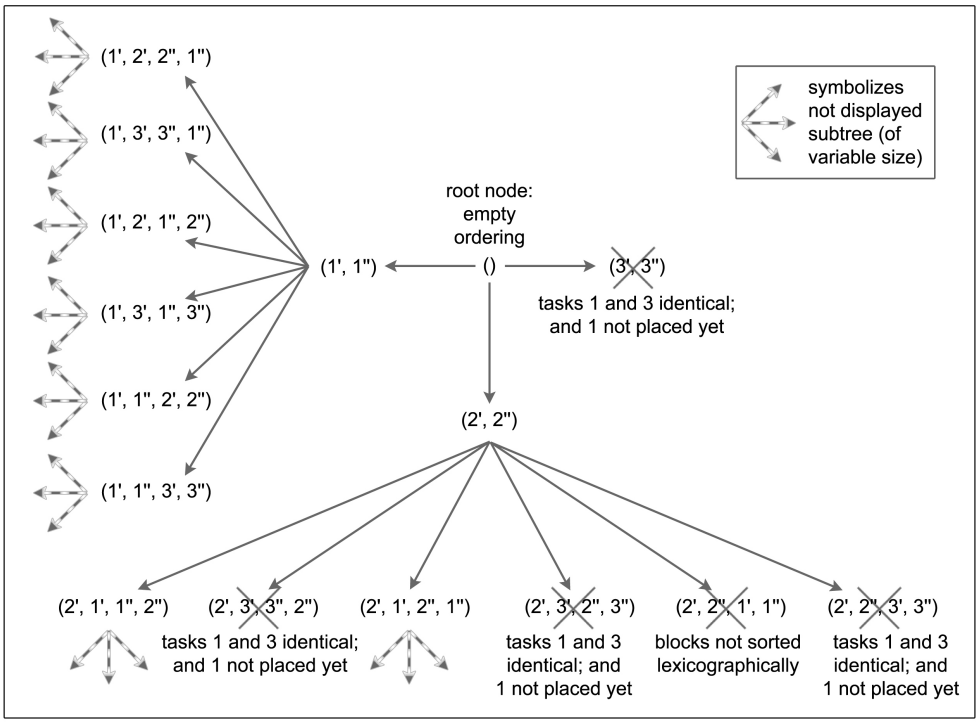

Figure 4: Generation of branch-and-bound tree and symmetry breaking.

presented in 3.3 we do not claim originality of the approaches. Time-indexed models or models with ordering variables have been presented also in [4, 9]. Many ideas are straightforward and we restrict ourselves here to short outlines stating the variables and basic constraints.

3.1 A time-indexed model

Time-indexed formulations are common for modeling machine scheduling problems since their linear programming relaxations often provide strong lower bounds. Their main disadvantage is, depending on the data, the quickly growing number of variables and constraints.

We use binary variables x_{it}, for $i = 1, 2, \ldots, n$ and $t = 0, 1, \ldots, T_{\max} - 1$ (for some upper bound $T_{\max}$ on the makespan) where x_{it} is 1 if job i is started at time t, and 0 otherwise.

Instead of modeling the problem as a min-max problem, we conduct a binary search for the optimum schedule length and try for each estimated makespan T to schedule as many jobs as possible within $[0, T]$. Since T might be too small, it is now only demanded that every job is scheduled at most once, i.e.,

$$\sum_{t=0}^{T-1} x_{it} \leq 1, \quad i = 1, 2, \ldots, n.$$

We introduce auxiliary binary variables z_{it}, for $i = 1, 2, \ldots, n$ and $t = 0, 1, \ldots, T$ with the interpretation that $z_{it} = 1$ if a subtask of job i occupies the machine in time interval $[t, t+1]$,

and $z_{it} = 0$ otherwise. The z- and x-variables are related by

$$z_{it} = \sum_{s=t-a_i+1}^{t} x_{is} + \sum_{s=t-t_i+1}^{t-a_i-l_i} x_{is},$$

for $0 \leq t \leq T$, where variables x_{is} with $s < 0$ are ignored.

The requirement that the execution of no two subtasks overlap, can simply be written as

$$\sum_{i=1}^{n} z_{it} \leq 1, \text{ for } t = 0, 1, \ldots, T-1.$$

For the IP-model we do not generate the auxiliary variables z_{it} but write these constraints directly in terms of the x_{it} variables.

We obtain a linear 0/1-programming formulation where the feasible solutions correspond to schedules of subsets of jobs in the interval $[0, T]$. The goal to schedule as many jobs as possible leads to the objective function

$$\max \sum_{i=1}^{n} \sum_{t=0}^{T-1} x_{it}.$$

3.2 A formulation with linear ordering variables

A basic requirement for the feasibility of a schedule is that all tasks are linearly ordered. To formulate this we introduce *linear ordering variables* x_{ij} for each pair of tasks i and j, $1 \leq i, j \leq 2n, i \neq j$, where $x_{ij} = 1$ if task i is started before task j, and $x_{ij} = 0$ otherwise. Clearly, the equation $x_{ij} + x_{ji} = 1$ has to be satisfied for $1 \leq i < j \leq 2n$. Furthermore, since for every job i its first task must be executed before its second task, we can fix the variables $x_{i_A i_B} = 1$, for $i = 1, \ldots, n$.

For enforcing a sequential ordering of the tasks we use the so-called *3-dicycle inequalities*

$$x_{ij} + x_{jk} + x_{ki} \leq 2$$

for every triple i, j, k of distinct tasks.

The linear ordering variables only guarantee that the tasks form a sequence. Of course, in addition the processing times and gap sizes have to be respected. For every job i we introduce variables s_{i_A} and s_{i_B} denoting the start times of its tasks. These variables can be defined as continuous variables because due to the integral data they will be integral in the optimal solution. Clearly, the relation $s_{i_B} = s_{i_A} + a_i + l_i$ holds.

Let U be an upper bound on the makespan. With the following constraints we model the differences between start times of pairs of tasks depending on the ordering.

(i) Difference between first task of job i and first task of job j

$$\begin{aligned} s_{i_A} &\leq U - a_i - t_j x_{i_A j_A}, \\ s_{j_A} &\leq U - a_j - t_i x_{j_A i_A}. \end{aligned}$$

(ii) Difference between first task of job i and second task of job j

$$\begin{aligned} s_{j_B} &\geq s_{i_A} + a_i - U(1 - x_{i_A j_B}), \\ s_{i_A} &\geq s_{j_B} + b_j - U(1 - x_{j_B i_A}). \end{aligned}$$

(iii) Difference between second task of job i and second task of job j

$$\begin{aligned} s_{j_B} &\geq s_{i_B} + b_i - U(1 - x_{i_B j_B}), \\ s_{i_B} &\geq s_{j_B} + b_j - U(1 - x_{j_B i_B}). \end{aligned}$$

The collection of the constraints given so far provides a linear integer model of the general coupled task problem. The objective function can be set up in a simple way. We add an extra job as new dummy job $n+1$ of length 0 and fix its respective linear ordering variables such that it can only be started after all tasks of the regular jobs have been executed. Then the objective function is just to minimize its starting time. Note that this is the same as introducing a variable T for the makespan to be minimized and requiring that it is greater than or equal to the completion time of any job.

We have conducted many experiments with sets of further constraints for strengthening the model depending on the concrete input data. For example, one can exclude some variable settings if processing times are too big for breaks, limit the total sum of processing times scheduled in the breaks of jobs, analyze the interdependence between jobs or incorporate optimum makespans computed for small sets of jobs. We do not go into further detail here, because computations did in general not reveal particular advantages when introducing these constraints.

3.3 An alternative formulation with linear ordering variables

This formulation also uses linear ordering variables, but there are no variables for start or termination times. Instead we introduce variables for idle times following the execution of tasks.

First note, that for a task k the value of $x_{ki_B} - x_{ki_A}$ is 1 if k is executed in the gap of job i, and 0 otherwise. We can use this expression to add for every job i the knapsack constraint

$$\sum_{k \notin \{i_A, i_B\}} (x_{ki_B} - x_{ki_A}) p_k \leq l_i.$$

stating that the sum of the processing times of tasks that are executed in the break of i must not exceed l_i.

The ordering and knapsack constraints together with the integrality of the x_{ij}-variables do not yet provide a characterization of orderings associated with feasible coupled task schedules. Consider the coupled task problem with three jobs where all tasks have length 1 and the breaks are $l_1 = l_3 = 2$ and $l_2 = 3$. Now take the ordering $\langle 1_A, 2_A, 3_A, 1_B, 2_B, 3_B \rangle$ and the corresponding setting of the x_{ij}-variables. All knapsack constraints are fullfilled since there are exactly two units of processing time used in each break. But there is only one feasible schedule which forces the break of job 2 to be exactly 2 (but it should be 3).

To overcome this problem, we introduce additional variables y_k, for $k = 1, \ldots, 2n$, giving the idle time of the processor after the execution of task k. The makespan can now be written as $\sum_{i=1}^{n}(a_i + y_{i_A} + b_i + y_{i_B})$ and, since the processing times are constant, the objective function becomes

$$\min \sum_{i=1}^{n} (y_{i_A} + y_{i_B}).$$

The idle time variables y_i have to be linked with the linear ordering variables x_{ij}. Let $\mathcal{G}_i$ be the set of tasks that are executed in the break of job i. Then, we can add all processing and idle times in this break and must obtain its length l_i. Therefore we have the equation

$$y_{i_A} + \sum_{j \in \mathcal{G}_i} (p_j + y_j) = l_i.$$

Since task j is in $\mathcal{G}_i$ if and only if $x_{ji_B} - x_{ji_A} = 1$ we obtain the equivalent equation

$$y_{i_A} + \sum_{j \notin \{i_A, i_B\}} (x_{ji_B} - x_{ji_A})(p_j + y_j) = l_i.$$

However, this is a quadratic equation and we apply the well-known transformation to linearize it by introducing new variables y_{ij}, $1 \leq i \leq n$, $1 \leq j \leq 2n$, $j \notin \{i_A, i_B\}$, for the products $(x_{ji_B} - x_{ji_A})y_j$. To ensure that $y_{ij} = y_j$ if $x_{ji_B} - x_{ji_A} = 1$ and $y_{ij} = 0$ otherwise, we add the constraints

$$\begin{aligned}
y_{ij} &\leq y_j, && 1 \leq i \leq n, 1 \leq j \leq 2n, j \notin \{i_A, i_B\}, \\
y_{ij} &\leq C_{ij}(x_{ji_B} - x_{ji_A}), && 1 \leq i \leq n, 1 \leq j \leq 2n, j \notin \{i_A, i_B\}, \\
y_{ij} &\geq y_j - C_j(x_{i_B j} + x_{ji_A}), && 1 \leq i \leq n, 1 \leq j \leq 2n, j \notin \{i_A, i_B\}.
\end{aligned}$$

The positive constants C_{ij} are upper bounds on the values of y_{ij} and $C_{ij} = \max\{0, l_i - p_j\}$ are suitable values. The upper bounds C_j for the values of y_j can be chosen as

$$C_j = \begin{cases} l_j, & \text{if } j \text{ is the first task of a job,} \\ \max_i\{C_{ij} \mid j \notin \{i_A, i_B\}\}, & \text{if } j \text{ is the second task of a job.} \end{cases}$$

The knapsack constraints can be now rewritten as linear equations

$$y_{i_A} + \sum_{j \notin \{i_A, i_B\}} \big((x_{ji_B} - x_{ji_A})p_j + y_{ij}\big) = l_i, \quad 1 \leq i \leq n.$$

3.4 The model of Sherali and Smith

Finally, we discuss one of the models which have been introduced in [9]. In contrast to the linear ordering models, it treats each job as one unit and does not consider the two tasks separately. To distinguish the relative processing order of the tasks of each pair of jobs i and j the following five cases are defined:

(i) Job i is finished before job j begins.

(ii) The four tasks of the jobs i and j are processed alternately, starting with the first task of job i.

(iii) Either job i is processed completely in the break of job j, or vice versa (at most one case is possible which can be read in advance from the concrete problem data).

(iv) The four tasks of the jobs i and j are processed alternately, starting with the first task of job j.

(v) Job j is finished before job i is started.

For each pair of jobs i, j and each case k, $1 \leq k \leq 5$, a binary variable y_{ijk} is introduced which equals one if and only if the corresponding configuration is realized. Furthermore, variables s_i for the start time of job i and a variable T for the makespan are introduced.

An integer model can then be formulated as

$$\begin{aligned}
\min_{s,y,T} \quad & T \\
\text{s.t.} \quad & s_j - s_i \geq \sum_{k=1}^{5} l_{ijk} y_{ijk}, \quad 1 \leq i < j \leq n, \\
& s_j - s_i \leq \sum_{k=1}^{5} r_{ijk} y_{ijk}, \quad 1 \leq i < j \leq n, \\
& T \geq s_i + t_i, \qquad 1 \leq i \leq n, \\
& \sum_{k=1}^{5} y_{ijk} = 1, \qquad 1 \leq i < j \leq n,
\end{aligned}$$

where l_{ijk} and r_{ijk} are constants depending on the concrete data of the jobs i and j as well as on the case k (for details see [9]).

4 Computational experiments

We studied the performance of the above approaches on different types of instances of the coupled task problem. For all models of Sections 3.1–3.4 an integer program was generated for each instance and solved with Cplex 12.1 using standard parameter settings. Note that for the time-indexed model, in addition a binary search has to be conducted. Therefore, a series of MIPs has to be solved for each instance. The branch-and-bound algorithm of section 2 is implemented in `C++`. Its implementation only uses one core of the machine.

The experiments were run on a PC with an Intel Quad Core CPU, 3.4GHz, 3Mb Cache and 16GB RAM. For every problem instance, the CPU time was limited to at most one hour. Prior to optimization a first-fit heuristic was called 10.000 times with random permutations of the jobs for obtaining a first feasible schedule and an upper bound on the optimum. CPU time for this heuristic was neglible and is not included.

We generated four types of random instances with the following specifications:

T1 All numbers a_i, l_i and b_i are uniformly distributed between 1 and 10.

T2 The breaks l_i are uniformly distributed between 1 and 10, all processing times a_i and b_i are equal to 1. (These problems are called *unit execution time* or *UET problems*. They have been considered in [1] for deriving worst-case bounds for approximation algorithms.)

T3 Data is generated according to [9] using normal distributions with mean value 30 and standard deviation 5 for a_i, mean value 600 and standard deviation 100 for l_i, and mean value 120 and standard deviation 30 for b_i.

T4 The processing times a_i and b_i are all uniformly distributed between 1 and 100 and the problems are generated in such a way that the tasks fit together almost perfectly in the sense that the optimum makespan exceeds the trivial lower bound $\sum(a_i + b_i)$ by at most 1 time unit.

For each of these types and for all problem sizes 10 random instances were generated. The results from our experiments are summarized in Tables 1 and 2. Each row gives the mean values for 10 randomly generated instances of the respective type and problem size. For the integer models the relative gaps between the lower bound at termination and the optimum solution value are given in percent, but the average is only taken over instances not solved to optimality. Furthermore the number of nodes of the branch-and-bound trees is given. The last column gives the average CPU times for the solved instances (for the other ones the time limit of one hour is reached).

		B&B		LOP model 1			LOP model 2		
	Size	Nodes	CPU	Gap	Nodes	CPU	Gap	Nodes	CPU
T1	8	2.66e2	0.0	0.00	3.58e3	0.6	0.00	7.38e2	0.5
	9	6.71e2	0.0	0.00	2.18e4	5.3	0.00	3.03e3	3.3
	10	2.27e3	0.0	0.00	9.04e4	21.7	0.00	1.11e4	14.9
	11	4.32e3	0.0	0.00	3.82e5	130.8	0.00	1.73e4	35.3
	12	1.14e4	0.0	0.00	3.18e6	1254.6	0.00	4.26e4	145.4
	20	1.93e7	104.2	40.29	2.69e5	3600.7	2.07	2.19e5	3241.9
T2	7	2.78e2	0.0	0.00	1.20e3	0.3	0.00	9.15e2	0.3
	8	8.14e2	0.0	0.00	1.12e4	2.5	0.00	5.54e3	2.3
	9	9.91e3	0.1	0.00	9.47e4	35.3	0.00	5.25e4	34.1
	10	3.42e4	0.2	0.00	5.64e5	306.6	0.00	1.68e5	157.9
	11	3.16e5	2.0	6.51	2.06e6	2787.8	2.00	1.03e6	1805.2
	12	1.49e6	11.3	16.17	1.03e6	3600.2	2.65	8.79e5	2521.4
T3	7	5.24e2	0.0	0.00	2.28e2	0.1	0.00	6.14e2	0.3
	8	2.73e3	0.0	0.00	1.52e3	0.7	0.00	6.11e3	3.6
	9	8.93e3	0.1	0.00	1.35e4	8.7	0.00	4.01e4	36.9
	10	3.28e4	0.5	0.00	1.42e5	157.8	0.00	3.06e5	580.4
	11	1.08e5	2.0	5.53	7.72e5	2721.0	2.66	8.58e5	3118.7
	12	4.58e5	8.9	20.82	8.81e5	3600.3	7.45	9.03e5	3600.2
T4	7	9.17e1	0.0	0.00	2.08e2	0.2	0.00	2.42e2	0.2
	8	2.86e2	0.0	0.00	1.13e3	0.7	0.00	1.28e3	1.5
	9	9.15e2	0.0	0.00	2.74e3	1.9	0.00	3.66e3	7.4
	10	3.40e3	0.0	0.00	1.87e4	19.0	0.00	1.37e4	37.8
	11	1.39e4	0.2	0.00	5.36e4	107.1	0.00	3.48e4	241.3
	12	4.33e4	0.8	0.00	1.29e5	568.9	0.55	1.05e5	1079.2

Table 1: Summary of comparison of the five approaches (1).

The time-indexed model is only suited for solving T2 problems. Here, it even outperforms the branch-and-bound algorithm. This is due to the fact, that the schedule is rather short and thus the size of the integer program is small. For some experiments analyzing the worst-case behaviour of certain approximative algorithms for UET problems we could even go to n beyond 100. For the other problem types, the makespans become too large, which strongly

		B&B		Time-index model			Sherali-Smith model		
	Size	Nodes	CPU	Gap	Nodes	CPU	Gap	Nodes	CPU
T1	8	2.66e2	0.0	0.00	2.21e4	22.5	0.00	1.31e4	0.5
	9	6.71e2	0.0	0.00	9.73e4	118.5	0.00	4.45e4	1.5
	10	2.27e3	0.0	0.00	4.00e5	647.9	0.00	2.40e5	6.3
	11	4.32e3	0.0	0.74	6.42e5	1830.8	0.00	8.67e5	23.8
	12	1.14e4	0.0	1.35	9.17e5	2431.4	0.00	5.71e6	200.2
	20	1.93e7	104.2	5.98	1.40e5	3600.0	52.56	1.11e7	3600.0
T2	7	2.78e2	0.0	0.00	4.00e-1	0.0	0.00	9.92e3	0.3
	8	8.14e2	0.0	0.00	5.73e1	0.1	0.00	7.43e4	2.2
	9	9.91e3	0.1	0.00	3.98e1	0.1	0.00	9.24e5	27.3
	10	3.42e4	0.2	0.00	3.27e2	0.1	0.00	1.11e7	382.9
	11	3.16e5	2.0	0.00	1.38e4	0.8	4.60	3.89e7	2426.5
	12	1.49e6	11.3	0.00	2.90e4	1.8	17.13	3.56e7	3600.0
T3	7	5.24e2	0.0	0.00	1.60e1	124.6	0.00	3.56e3	0.1
	8	2.73e3	0.0	0.91	7.42e2	1297.0	0.00	4.62e4	1.2
	9	8.93e3	0.1	1.86	5.28e2	3166.5	0.00	5.30e5	14.0
	10	3.28e4	0.5	6.42	4.50e2	3600.1	0.00	5.46e6	197.9
	11	1.08e5	2.0	5.60	2.34e2	3600.1	0.81	3.09e7	1785.0
	12	4.58e5	8.9	9.43	2.52e2	3600.2	7.25	3.47e7	2946.6
T4	7	9.17e1	0.0	0.00	0.00	5.9	0.00	3.47e3	0.1
	8	2.86e2	0.0	0.00	1.28e1	48.5	0.00	1.01e4	0.5
	9	9.15e2	0.0	0.00	0.00	17.3	0.00	2.21e4	1.2
	10	3.40e3	0.0	3.35	1.65e3	2228.3	0.00	1.99e5	8.9
	11	1.39e4	0.2	5.12	1.81e3	3256.7	0.00	4.23e5	29.7
	12	4.33e4	0.8	7.64	1.31e3	3600.1	0.00	7.70e5	105.0

Table 2: Summary of comparison of the five approaches (2).

increases the problem size and the computation of optimum solutions is out of reach for this approach.

Comparing the results of the two linear ordering based models and the Sherali-Smith model, we see that the Sherali-Smith model can solve the problems faster to optimality if they can be solved within the time limit. However, for the problems which could not be solved within the time limit, the gap of at least one of the linear ordering models is generally smaller, which is due to better lower bound values. Hence, we can conclude that the linear ordering formulations provide tighter root node relaxations but are slower in closing the gap because the solution times for the linear programming relaxations are much larger. For T2, T3 and T4 problems these models clearly outperform the time-indexed model.

The branch-and-bound algorithm compares favorably with all other formulations. All instances in the testset of Tables 1 and 2 could be solved within the time limit and with a maximum solution time of 415 seconds. Generally, both the number of branch-and-bound nodes needed and the computing time per node, are much smaller for this algorithm than for the others.

We explored the competitiveness of the time-index model on T2 instances further by considering larger instances. At first glance, the times do not seem to be that much better. But, the branch-and-bound algorithm only solves those instances where the optimum makespan equals the trivial bound $\sum_i a_i + b_i$. Here the solution is always found in the first couple of hundred nodes with solution times much less than 1 second. None of the non-trivial so-

	B&B				Time-index model			
Size	Sol	CPU	Nodes	Gap	Sol	CPU	Nodes	Gap
13	10	81	1.24e+07	0.0	10	17	1.36e+05	0.0
14	10	745	9.86e+07	0.0	10	121	9.18e+05	0.0
15	6	0	2.08e+08	3.2	10	424	2.37e+06	0.0
16	4	0	2.31e+08	3.0	7	442	1.27e+07	3.0

Table 3: Comparison between branch-and-bound and time-indexed model on T2 instances.

lutions of the problems with size 15 or 16 are found by branch-and-bound within the time limit. Here, the effect of the bounding procedures described in Section 2 is weak and hence branch-and-bound generates very many nodes.

5 Conclusions

Our computational results show that the branch-and-bound approach outperforms the other models by far, except when the processing times of the jobs a_i and b_i are very small. Then, the bounding heuristics usually do not improve the bound at all and a lot of time is wasted. In this case, the time-indexed model is clearly the best choice, because the number of variables remains fairly small.

The effectiveness of the other models is highly depending on the structure of the problem instance to be solved. E.g., we considered a coupled-task problem instance with identical jobs, where $n = 9$, $a_i = 3$, $l_i = 6$, $b_i = 4$. The time-indexed model solved this problem in about 1 second with 9 subproblems and a root bound of 78. The Sherali-Smith model was stopped after 1 hour after having generated more than 800 000 nodes with a lower bound of still 55. However, for this problem type, neither of our approaches is appropriate, since there exists a specially tailored dynamic programming algorithm given in [2] which can easily solve such problem instances even for n up to 1000.

From the study of the coupled task problem we conclude that, although integer linear programming approaches might seem more elegant, tailored branch-and-bound algorithms (and sometimes even dynamic programming algorithms) should not be overlooked. Depending on the structure of the problem and the size of the instances they can be much more effective. Furthermore, branch-and-bound has the additional advantage that it usually can be parallelized in a fairly straightforward way and even make use of massively parallel hardware.

Acknowledgement

We would like to thank two anonymous referees for very helpful comments and suggestions for improving a first version of this paper.

References

[1] A.A. Ageev and A.E. Baburin: Approximation algorithms for UET scheduling problems with exact delays, *Operations Research Letters* **25**, 533–540, 2007.

[2] D. Ahr, J. Békési, G. Galambos, M. Oswald, and G. Reinelt: An Exact Algorithm for Scheduling Identical Coupled Tasks, *Mathematical Methods of Operations Research* **59**, 193–203, 2004.

[3] J. Blazewicz, K. Ecker, T. Kis, and M. Tanas: A Note on the Complexity of Scheduling Coupled Tasks on a Single Processor, *Journal of the Brazilian Computer Society* **7**, 23–26, 2001.

[4] M. Elshafei, H.D. Sherali, and J.C. Smith: Radar pulse interleaving for multi-target tracking, *Naval Research Logistics* **51 (4)**, 72–94, 2004.

[5] H. Li and H. Zhao: Scheduling Coupled-Tasks on a Single Machine, in: *IEEE Symposium on Computational Intelligence in Scheduling SCIS '07*, 137–142, 2007.

[6] A.J. Orman and C.N. Potts: On the Complexity of Coupled-Task Scheduling, *Discrete Applied Mathematics* **72**, 141–154, 1997.

[7] A.J. Orman, C.N. Potts, A.K. Shahani, and A.R. Moore: Scheduling for a multifunction phased array radar system, *European Journal of Operational Research* **90**, 13–25, 1996.

[8] C.N. Potts and J.D. Whitehead: Heuristics for a Coupled-Operation Scheduling Problem, *The Journal of the Operational Research Society* **58**, 1375–1388, 2007.

[9] H.D. Sherali and J.C. Smith: Interleaving Two-Phased Jobs on a Single Machine with Application to Radar Pulse Interleaving, *Discrete Optimization* **2**, 348–361, 2005.

[10] M. Tanas, J. Blazewicz, and K. Ecker: Polynomial Time Algorithm for Coupled Tasks Scheduling Problem, in: J. Blazewicz, K. Ecker and B. Hammer (eds): ICOLE 2007, Lessach, Austria, Report IfI-07-03, TU Clausthal, 76–69, 2007.